Backendpy

Release 0.1.7a1

Jalil Hamdollahi Oskouei

Dec 05, 2022

CONTENTS:

CHAPTER

ONE

INTRODUCTION

Backendpy is an open-source framework for building the back-end of web projects with the Python programming language.

1.1 Why Backendpy?

This framework does not deprive developers of their freedom by restricting them to pre-defined structures, nor does it leave some repetitive and time-consuming tasks to the developer.

Some of the features of Backendpy are:

- Asynchronous programming (ASGI-based projects)
- Application-based architecture and the ability to install third-party applications in a project
- Support of middlewares for different layers such as Application, Handler, Request or Response
- Supports events and hooks
- Data handler classes, including validators and filters to automatically apply to request input data
- Supports a variety of responses including JSON, HTML, file and… with various settings such as stream, gzip and…
- Router with the ability to define urls as Python decorator or as separate files
- Application-specific error codes
- Optional default database layer by the Sqlalchemy async ORM with management of sessions for the scope of each request
- Optional default templating layer by the Jinja template engine
- …

1.2 License

The Backendpy framework licensed under the BSD 3-Clause terms. The source code is available at https://github.com/savangco/backendpy.

Note: This project is under active development.

CHAPTER

TWO

INSTALLATION

2.1 Requirements

Python 3.8+

2.2 Using pip

To install the Backendpy framework using pip:

```
$ pip3 install backendpy
```

If you also want to use the optional framework features (such as database, templating, etc.), you can use the following command to install the framework with additional dependencies:

```
$ pip3 install backendpy[full]
```

If you only need one of these features, you can install the required dependencies separately. The list of these requirements is as follows:

Table 1: Optional requirements

Name	Version	Usage
asyncpg	>=0.25.0	If using default ORM
sqlalchemy	>=1.4.27	If using default ORM
jinja2	>=3.0.0	If using default Templating
aiohttp	>=3.8.0	If using the AsyncHttpClient class of the backendpy.utils.http
requests	>=2.27.0	If using the HttpClient class of the backendpy.utils.http
pillow	>=9.0.0	If using the ModifyImage filter of the backendpy.data_handler.filters
ujson	>=5.1.0	If installed, ujson.loads will be used instead of json.loads, which is faster

You also need to install an ASGI server such as Uvicorn, Hypercorn or Daphne:

```
$ pip3 install uvicorn
```

CHAPTER

THREE

START A PROJECT

3.1 Create a project

3.1.1 Basic structure

A Backendpy-based project does not have a mandatory, predetermined structure, and it is the programmer who decides how to structure his project according to his needs.

The programmer only needs to create a Python module with a custom name (for example "main.py") and set the instance of `Backendpy` class (which is an ASGI application) inside it.

Listing 1: project/main.py

```
from backendpy import Backendpy

bp = Backendpy()
```

Also for project settings, the `config.ini` file must be created in the same path next to the module. Check out the *Configurations* section for more information.

3.1.2 Applications

Backendpy projects are developed by components called Applications. It is also possible to connect third-party apps to the project.

To create an application, first create a package containing the `main.py` module in the desired path within the project (or any other path that can be imported).

Then inside the main.py module of an application we need to set an instance of the `App` class. All parts and settings of an application are assigned by the parameters of the App class.

For example, in the "/apps" path inside the project, we create a package called "hello" and main.py file as follows:

Listing 2: project/apps/hello/main.py

```
from backendpy.app import App
from .handlers import routes

app = App(
    routes=[routes])
```

Listing 3: project/apps/hello/handlers.py

```
from backendpy.router import Routes
from backendpy.response import Text

routes = Routes()

@routes.get(r'^/hello-world$')
async def hello_world(request):
    return Text('Hello World!')
```

As you can see, we have created another optional module called handlers.py and then introduced the routes defined in it to the App class instance. The complete list of App class parameters is described in section *Applications structure*.

Only the items that are introduced to the App class are important to the framework, and the internal structuring of the applications is completely optional.

Our application is now ready and you just need to enable it in the project config.ini file as follows:

Listing 4: project/config.ini

```
[apps]
active =
    project.apps.hello
```

To run the project, see the *Run* section.

Refer to the *Application development* section to learn how to develop applications.

3.1.3 Command line

The backendpy command can also be used to create projects and apps. To do this, first enter the desired path and then use the following commands:

Project creation

```
$ backendpy create_project --name myproject
```

To create a project with more complete sample components:

```
$ backendpy create_project --name myproject --full
```

App creation

```
$ backendpy create_app --name myapp
```

```
$ backendpy create_app --name myapp --full
```

3.2 Run

You can use different ASGI servers such as Uvicorn, Hypercorn and Daphne to run the project. For this purpose, you must first install your desired server (see the *Installation* section).

Then enter the project path and use the following commands:

Listing 5: For Uvicorn

```
$ uvicorn main:bp
```

Listing 6: For Hypercorn

```
$ hypercorn main:bp
```

Listing 7: For Daphne

```
$ daphne main:bp
```

Note: In these examples, we assume that the name of the main module of the project is "main.py" and the instance name of the Backendpy class inside it is "bp". These names are optional.

The server is now accessible (depending on the host and port running on it) for example at http://127.0.0.1:8000.

For more information on the options of each server, refer to their documentation.

3.3 Configurations

In Backendpy projects, all the settings of a project are defined in the `config.ini` file, which is located in the root path of each project and next to the main module of the project. This config file is defined in INI format, which includes sections and options. The basic list of framework configs and example of their definition is as follows:

Listing 8: project/config.ini

```
; Backendpy Configurations

[networking]
allowed_hosts =
    127.0.0.1:8000
    localhost:8000
stream_size = 32768

[environment]
media_path = /foo/bar

[apps]
active =
    backendpy_accounts
    myproject.apps.myapp

[middlewares]
```

(continues on next page)

(continued from previous page)

```
active =
    backendpy_accounts.middleware.auth.AuthMiddleware

[database]
host = localhost
port = 5432
name =
username =
password =
```

In ini format, `;` is used for comments, `[]` is used to define sections, `key = value` is used to define values and the lines are used for list values.

- **networking** section contains values related to the server and the network.
- **environment** section contains values such as the path to the media files and etc.
- **apps** section contains a list of the project active applications.
- **middlewares** section contains a list of the project active middlewares.
- **database** section, if using the default ORM, will include the settings related to it.

Also other custom settings may be required by any of the active apps, which must also be specified in this file. For example, an account application might have settings like this:

```
[keys]
aes_key = 11111111111111111111111111111111
auth_tokens_secret = 22222222222222222222222222
```

Note: To protect sensitive information in the config file, such as passwords, private keys, etc., be sure to restrict access to this file. For example, set the permission to 600.

In order to access the project configs inside the code, you can use the `config` attribute of the project `Backendpy` class instance which contains this configs in dictionary format:

Listing 9: project/main.py

```
from backendpy import Backendpy

bp = Backendpy()

print(bp.config['database']['host'])
```

And similarly inside the request handlers:

Listing 10: project/apps/hello/handlers.py

```
async def hello_world(request):
    print(request.app.config['database']['host'])
    ...
```

3.4 Environments

In Backendpy framework, it is possible to use different environments for project execution (such as development, production, etc.).

Each environment has a different config file and can have different database settings, media paths, and so on.

In a development team, different parts of the team can use their own environments and settings to execute and work on the project. Also, if needed on the main server, the project can be executed with another configuration in parallel, on another host or port.

To do this, you need to define the `BACKENDPY_ENV` variable in the os with the desired name for the environment. For example, to define an environment called "dev", we use the following command:

```
$ export BACKENDPY_ENV=dev
```

You must also create and configure a separate configuration file named `config.dev.ini`.

Now if the server runs (in the process that the environmental variable is exported) or when other backendpy management commands are executed, this configuration will be used instead of the original configuration.

3.5 Management

The following commands can be used by the system administrator:

3.5.1 Creating Database

If you are using the default ORM, you can create the database and all data model tables within the project automatically by the following command (after entering the project path in the command line):

```
$ backendpy create_db
```

3.5.2 Initialization

Some applications require the initial storage of data in the database, the creation of files, and so on, before they can be used. For example, before using some systems, information such as user roles, admin account, etc. related to the users application must be stored in the database.

By running the following command in the project path, Backendpy framework will execute the initialization scripts of all the apps enabled on the project and will also take the required input data on the command line:

```
$ backendpy init_project
```

CHAPTER

FOUR

APPLICATION DEVELOPMENT

4.1 Applications structure

In section *Create a project*, we talked about how to create and activate a basic application in a Backendpy-based project. In this section, we describe the complete components of an application.

As mentioned earlier, a Backendpy-based application does not have a predefined structure or constraint. In fact, the developer is free to implement the desired architecture for the application and finally import and configure all the components inside the main module of the application.

The main module of an application must contain an instance of the `App` class that is defined inside a variable called `app`. Below is an example of defining an app with all its possible parameters (which are used to assign components to an application):

Listing 1: project/apps/hello/main.py

```
from backendpy.app import App
from .controllers.handlers import routes
from .controllers.hooks import hooks
from .controllers.errors import errors
from .controllers.init import init_func

app = App(
    routes=[routes],
    hooks=[hooks],
    errors=[errors],
    models=['project.apps.hello.db.models'],
    template_dirs=['templates'],
    init_func=init_func)
```

An `App` class has the following parameters:

In the following, we will describe each of these components of the application, as well as other items that can be used in applications.

4.2 Routes

Routes in an application are the accessible URLs defined for that application. The routes of an application are defined according to a specific format, and for each route, a handler function is assigned. When each request reaches the server, if the request url matches with a route, the request will be delivered to the handler of that route.

Generally, there are two ways to define routes, one is to use Python decorators on top of handler functions, and the other is to use a separate section like urls.py file containing a list of all the routes and linking them to the handlers. For different developers and depending on the architecture of the application, each of these methods can take precedence over the other. One of the features of the Backendpy framework is the possibility of defining routes in both methods.

Consider the following examples:

4.2.1 Decorator based routes

To define `Uri` we can use `get()`, `post()`, `path()`, `put()` and `delete()` decorators as follows:

Listing 2: project/apps/hello/handlers.py

```
from backendpy.router import Routes
from backendpy.response import Text

routes = Routes()

@routes.get(r'^/hello-world$')
async def hello_world(request):
    return Text('Hello World!')

@routes.post(r'^/login$')
async def login(request):
    ...
```

Also, if we need to access a handler with different http methods, we can use `uri()` decorator as follows:

Listing 3: project/apps/hello/handlers.py

```
from backendpy.router import Routes
from backendpy.response import Text

routes = Routes()

@routes.uri(r'^/hello-world$', ('GET', 'POST'))
async def hello_world(request):
    return Text('Hello World!')
```

4.2.2 Separate routes

We can define the list of `Uri` separately from the handlers as follows:

Listing 4: project/apps/hello/handlers.py

```
from backendpy.response import Text

async def hello_world(request):
    return Text('Hello World!')

async def login(request):
    ...
```

Listing 5: project/apps/hello/urls.py

```
from backendpy.router import Routes, Uri
from .handlers import hello_world, login

routes = Routes(
    Uri(r'^/hello-world$', ['GET'], hello_world),
    Uri(r'^/login', ['POST'], login),
)
```

As can be seen in the examples, in both cases, the `Routes` object is defined, which is used to hold the list of `Uri`.

The complete list of parameters of a `Uri` is as follows:

Note that in `@uri` decorator, which is defined on the handler function itself, the `handler` parameter does not exist. and in `@get`,``@post`` and ... decorators, the `method` parameter also does not exist.

After defining the routes, the Routes object can then be assigned to the application via the `App` class `routes``parameter in the ``main.py` module of the application:

Listing 6: project/apps/hello/main.py

```
from backendpy.app import App
from .handlers import routes

app = App(
    routes=[routes])
```

In an application, more than one object of the Routes class can be defined. Each of which can be used to define the Uri of separate parts of the application or even different versions of the API and the like. For example:

Listing 7: project/apps/hello/main.py

```
from backendpy.app import App
from .handlers.v1 import routes as routes_v1
from .handlers.v2 import routes as routes_v2

app = App(
    routes=[routes_v1, routes_v2])
```

4.3 Requests

HTTP requests, after being received by the framework, become a `Request` object and are sent to the handler functions as a parameter called `request`.

Listing 8: project/apps/hello/handlers.py

```
async def hello_world(request):
    ...
```

Request object contains the following fields:

4.4 Responses

To respond to a request, we use instances of the `Response` class or its subclasses inside the handler function. Default Backendpy responses include `Text`, `HTML`, `JSON`, `Binary`, `File`, and `Redirect`, but you can also create your own custom response types by extending the `Response` class.

The details of each of the default response classes are as follows:

Example usage:

Listing 9: project/apps/hello/handlers.py

```
from backendpy.router import Routes
from backendpy.response import Text

routes = Routes()

@routes.get(r'^/hello-world$')
async def hello_world(request):
    return Text('Hello World!')
```

Example usage:

Listing 10: project/apps/hello/handlers.py

```
from backendpy.router import Routes
from backendpy.response import JSON

```

(continues on next page)

(continued from previous page)

```
routes = Routes()

@routes.get(r'^/hello-world$')
async def hello_world(request):
    return JSON({'message': 'Hello World!'})
```

Example usage:

Listing 11: project/apps/hello/handlers.py

```
from backendpy.router import Routes
from backendpy.response import HTML

routes = Routes()

@routes.get(r'^/hello-world$')
async def hello_world(request):
    return HTML('<html><body>Hello World!</body></html>')
```

Example usage:

Listing 12: project/apps/hello/handlers.py

```
import os
from backendpy.router import Routes
from backendpy.response import File

routes = Routes()

@routes.get(r'^/hello-world$')
async def hello_world(request):
    return File(os.path.join('images', 'logo.jpg'))
```

There is another type of response to quickly return a success response in predefined json format, which is as follows:

Example usage:

Listing 13: project/apps/hello/handlers.py

```
from backendpy.router import Routes
from backendpy.response import Success

routes = Routes()

@routes.get(r'^/hello-world$')
async def hello_world(request):
    return Success()

@routes.post(r'^/login$')
async def login(request):
    return Success('Successful login!')
```

Note: The json format used in the `Success` response is similar to the `Error` response, and these two types of responses

can be used together in a project. Refer to the *Predefined errors* section for information on how to use the `Error` response.

4.5 Exceptions

Backendpy exceptions are the type of responses used to return HTTP errors and can also be raised.

Note: As mentioned, Backendpy exceptions are the type of `Response` and their content is returned as a response and displayed to the user. Therefore, these exceptions should be used only for errors that must be displayed to users, and any kind of internal system error should be created with normal Python exceptions, in which case, the `ServerError` response is displayed to the user with a public message and does not contain sensitive system information that may be contained in the internal exception message.

The list of default exception response classes are as follows:

Example usage:

Listing 14: project/apps/hello/handlers.py

```
from backendpy.router import Routes
from backendpy.exception import BadRequest

routes = Routes()

@routes.post(r'^/login$')
async def login(request):
    raise BadRequest({'message': 'Login failed!'})
```

4.6 Predefined errors

To return a response with error content, you can manually return the content of that error with *Exceptions* or create helper classes and assign specific templates to them. But if the number and variety of these errors is large, it is better to assign a code to each error in addition to a fixed format for errors.

In the Backendpy framework, a special class for managing error responses is provided to help better and easier manage errors and their code and to prevent the error code of one application from interfering with other applications:

In order to use `Error` response class, you must first define a list of errors for the application, including codes and their corresponding messages. The list of errors in an application is defined by an instance of the `ErrorList` class, which itself contains instances of the `ErrorCode` class.

For example, inside a custom module:

Listing 15: project/apps/hello/controllers/errors.py

```
from backendpy.response import Status
from backendpy.error import ErrorCode, ErrorList

errors = ErrorList(
    ErrorCode(1100, "Authorization error", Status.UNAUTHORIZED),
```

(continues on next page)

(continued from previous page)

```
    ErrorCode(1101, "User '{}' does not exists", Status.BAD_REQUEST)
)
```

After defining the list of errors, we must add this list to the application. For this purpose, as mentioned before, inside the `main.py` module of the application, we set the `errors` parameter with our own error list (instance of `ErrorList` class).

Also note that the errors parameter is a list type, and more than one ErrorList can be assigned to each app, each list being specific to a different part of the app.

Listing 16: project/apps/hello/main.py

```
from backendpy.app import App
from .controllers.errors import errors

app = App(
    ...
    errors=[errors],
    ...)
```

And finally, examples of returning the `Error` response:

Listing 17: project/apps/hello/controllers/handlers.py

```
from backendpy.router import Routes
from backendpy.error import Error

routes = Routes()

@routes.get(r'^/example-error$')
async def example_error(request):
    raise Error(1100)

@routes.post(r'^/login$')
async def login(request):
    raise Error(1101, 'jalil')
```

4.7 Data handlers

One of the capabilities of Backendpy framework is the input data handlers, which includes default or user-defined validators and filters. With this feature, the main handlers of the requests and the handlers of their input data are separated, and in this case, instead of the raw data, the validated and filtered data are received inside the main handlers.

With the regular and reusable structure of Data Handlers, much of the need for duplicate coding as well as unrelated to the main logic within the code is eliminated and speeds up project implementation.

Data handlers are defined as classes that inherit from the `Data` class.

For example:

Listing 18: project/apps/hello/controllers/data_handlers.py

```
from backendpy.data_handler.data import Data
from backendpy.data_handler.fields import String
from backendpy.data_handler import validators as v
from backendpy.data_handler import filters as f
...

class UserCreationData(Data):
    group = String('group', required=True, processors=[v.NotNull()], field_type=TYPE_URL_
↪VARS)
    first_name = String('first_name', processors=[v.Length(max=50)])
    last_name = String('last_name', processors=[v.Length(max=50)])
    email = String('email', processors=[v.EmailAddress()])
    username = String('username', processors=[v.Unique(model=Users)])
    password = String('password', processors=[v.PasswordStrength()])
    password_re = String('password_re')
```

each of the items in the example is described below.

After defining a data handler, we must assign it to a request. This allocation is done in the routes definition section with the `data_handler` parameter:

Listing 19: project/apps/hello/controllers/handlers.py

```
from backendpy.router import Routes
from .data_handlers import UserCreationData

routes = Routes()

@routes.post(r'^/users$', data_handler=UserCreationData)
async def user_creation(request):
    data = request.cleaned_data
    ...
```

To get the final validated and filtered data inside the request main handler, we use `request.cleaned_data`, which will be a dictionary of data with defined fields in our data handler class.

4.7.1 Data fields

As shown in the previous example, data fields are defined inside the data handler class. Each field can be an instance of `Field` class or other data classes inherited from this base class.

In the example, `String` field is used. Developers can also create and use their own custom data fields as needed.

The parameters of the base field class are as follows:

4.7.2 Data processors

Processors are classes for processing data field values that include validators and filters.

A list of processors is assigned to a data field via the `processors` parameter and will run in sequence as specified. Also in this list, validators and filters can be used with any combination.

Validators

Validators are responsible for reviewing and validating data, and a data is either passed over or, if there is a discrepancy, the defined error is returned.

Developers can create and use the various validators they need by inheriting from the base `Validator` class.

Ready-made validators are also provided in the framework that can be used. The following is a list of them:

Default validators

Example:

```
token_type = String('token_type', required=True, processors=[v.NotNull(), v.In(['basic',
↪'bearer'])])
```

Example:

```
image = String('image', processors=[v.NotNull(), v.RestrictedFile(extensions=('jpg',
↪'jpeg', 'png'), min_size=1, max_size=2048)])
```

In this example, if the data we receive is a list of images instead of an image file, and we want these processors to be applied to all of those images, we can nest the list of processors inside another list as follows:

```
images = String('images', processors=[list((v.NotNull(), v.RestrictedFile(extensions=(
↪'jpg', 'jpeg', 'png'), min_size=1, max_size=2048)))])
```

Example:

```
username = String('username', processors=[v.Unique(model=Users)])
```

In this example, the value sent to the "username" field is queried directly to the "username" column from the "Users" model and checked for its uniqueness, and returns an error if it is exists.

In the previous example, if the name of the model table field is "user_id" instead of "username", we should change it as follows:

```
username = String('username', processors=[v.Unique(model=Users, model_field_name='user_id
↪')])
```

The previous example was for adding a new user with a unique username to the database; However, if our request is to edit a user, the previous example should change as follows to prevent the error from being displayed when the user's current username is resubmitted:

```
username = String('username', processors=[v.Unique(model=Users, model_field_name='user_id
↪', except_self='id')])
```

Note that in this example the "id" column of the model is used to identify each row of data. It is also necessary to send a field named "id" with the value of the current row id of this user in the database in the submitted data in order to exclude this row when checking the uniqueness of the username.

Filters

Filters are responsible for modifying data as needed, and changes are made when data passes through it.

Developers can create and use the various filters they need by inheriting from the base `Filter` class.

Default filters are also can be used:

Default filters

Example:

```
from backendpy.data_handler import validators as v
from backendpy.data_handler import filters as f
...

image = String('image', processors=(v.NotNull(), f.DecodeBase64(), v.
↪RestrictedFile(extensions=('jpg', 'jpeg', 'png'), min_size=1, max_size=2048), f.
↪ModifyImage(format='JPEG')))
```

In this example, a combination of validators and filters is used. First it checks that the value is not null, then it applies a filter to the received data and decodes it from base64 format, then it checks the allowed extensions for the received file with validator, and if it passes, it converts the file to jpeg format with another filter.

4.8 Hooks

Sometimes it is necessary to perform a specific operation following an event. For these types of needs, we can use Backendpy hooks feature. For example, when we want to write an email management application that sends an email after certain events in other applications, such as the registration or login of users.

With this feature, we can both define new events with special labels within our application as points so that others can write their own code for these events to run, or we can assign codes to execute when triggering other events on the system.

4.8.1 Event Definition

To define event points, we use the `execute_event` method of the `Backendpy` class instance inside any space we have access to this instance. (For example, inside the handler of a request, we access the project `Backendpy` instance via `request.app`).

Example of defining user creation event:

```
@routes.post(r'^/users$', data_handler=UserCreationData)
async def user_creation(request):
    ...
    await request.app.execute_event('user_created')
    ...
```

If the event also contains arguments, we send them in the second parameter in the form of a dictionary:

```
@routes.post(r'^/users$', data_handler=UserCreationData)
async def user_creation(request):
    ...
    await request.app.execute_event('user_created', {'username': username})
    ...
```

Default Events

In addition to the events that developers can add to the project, the default events are also provided in the framework as follows:

Table 1: Framework Default Events

Label	Description
startup	After successfully starting the ASGI server
shutdown	After the ASGI server shuts down
request_start	At the start of a request
request_end	After the response to a request is returned

4.8.2 Hook Definition

To define the code that is executed in events, we use the `Hooks` class and its `event` decorator:

Listing 20: project/apps/hello/controllers/hooks.py

```
from backendpy.hook import Hooks
from backendpy.logging import get_logger

LOGGER = get_logger(__name__)
hooks = Hooks()

@hooks.event('startup')
async def example():
    LOGGER.debug("Server starting")

@hooks.event('user_created')
async def example2(username):
    LOGGER.debug(f"User '{username}' creation")
```

As can be seen, if an argument is sent to a hook, these arguments are received in the parameters of the hook functions, otherwise they have no parameter.

Here we have written the hooks inside a custom module. To connect these hooks to the application, like the other components, we use the `main.py` module of the application:

Listing 21: project/apps/hello/main.py

```
from backendpy.app import App
from .controllers.hooks import hooks

app = App(
```

(continues on next page)

(continued from previous page)

```
    ...
    hooks=[hooks],
    ...)
```

Another way to use hooks is to attach them directly to a project (instead of an application), which can be used for special purposes such as managing database connections, which are part of the project-level settings:

Listing 22: project/main.py

```
from backendpy import Backendpy

bp = Backendpy()

@bp.event('startup')
async def on_startup():
    LOGGER.debug("Server starting")
```

4.9 Middlewares

An ASGI application based on the Backendpy framework (instance of `Backendpy` class) can be used with a variety of external ASGI middlewares. In addition to external middlewares, in the Backendpy framework itself, the ability to create middleware for the internal components and layers of the system is also available. These types of internal middlewares are discussed below:

4.9.1 Types of middleware

The types of Backendpy internal middlewares depending on the layer they are processing are as follows:

Backendpy instance middleware

This type of middleware, like the external middleware mentioned earlier, processes an ASGI application (instance of `Backendpy` class) and adds to or modifies its functionality.

The difference between this type of middleware and external middleware is the easier way to create and attach it to the project, which instead of changing the code, we set it in the project config file.

Request middleware

This middleware takes a `Request` object before it reaches the handler layer and delivers a processed or modified Request object to the handler layer.

Also, depending on the type of processing in this middleware, the middleware can prevent the request process from continuing and interrupt it with an error response and prevent it from reaching the handler layer.

Handler middleware

This middleware takes a request handler (which is an async function) before executing it and returns a processed or modified handler. As a result, this new handler will be used instead of the original handler to return the response to this request.

In this middleware, it is also possible to interrupt the process and return the error response.

Response middleware

This middleware captures the final `Response` object before sending it to the client and returns a processed, modified, or replaced Response object.

4.9.2 Creating middleware

To create a middleware, use `Middleware` class and implement its methods. Each of these methods is specific to implementing different types of middleware mentioned in the previous section.

How to define these methods is as follows:

Listing 23: project/apps/hello/middlewares/example.py

```
from backendpy.middleware import Middleware

class MyMiddleware(Middleware):

    @staticmethod
    def process_application(application):
        ...
        return application

    @staticmethod
    async def process_request(request):
        ...
        return request

    @staticmethod
    async def process_handler(request, handler):
        ...
        return handler

    @staticmethod
    async def process_response(request, response):
        ...
        return response
```

As can be seen, all methods are static and also except for `process_application` which is a simple function, all other methods (which are in the path of handling a request) must be defined as an `async` function.

As an example of a request middleware, it can be used to authenticate the user before executing the request handler:

Listing 24: project/apps/hello/middlewares/auth.py

```
from backendpy.middleware import Middleware
from backendpy.exception import BadRequest
...

class AuthMiddleware(Middleware):

    @staticmethod
    async def process_request(request):
        auth_token = request.headers.get('authorization')

        if is_invalid_token(auth_token):
            raise BadRequest({'error': 'Invalid auth token'})

        user = get_user(auth_token)

        request.context["auth"] = {
            'user_id': user.id,
            'user_roles': user.roles}

        return request
```

In this example, after receiving a request, first the user identity is checked inside the middleware and if there is an error, the error response is returned and if successful, the user information is added to the request context and we can access this information inside the request handler.

4.9.3 Activation of middleware

In order to activate a middleware in a project, we need to define them in the project `config.ini` file as follows:

Listing 25: project/config.ini

```
...
[middlewares]
active =
    project.apps.hello.middlewares.auth.AuthMiddleware
```

The middlewares are independent classes and can be written as part of an application or as a standalone module. In both cases, to enable them, their class must be added to the project config. This means that by activating an application, its internal middlewares will not be enabled by default.

Note: Note that in a project you can define an unlimited number of middlewares of one type or in different types. Middlewares of the same type will be queued and executed in the order in which they are defined, and the output of each middleware will be passed to the next middleware.

4.10 Database

In Backendpy, developers can use any database system of their choice, depending on the needs of the project, using any type of engine layer that supports async requests, so there is no mandatory structure for this. However, this framework will provide helpers to speed up work with various database systems. Helpers are currently available for Sqlalchemy.

4.10.1 Use custom database

With the *Hooks* feature described in the previous sections, you can easily configure your connections, sessions, and database system according to different system events.

For example:

Listing 26: project/main.py

```
from backendpy import Backendpy
...

bp = Backendpy()

@bp.event('startup')
async def on_startup():
    bp.context['db_engine'] = get_db_engine(config=bp.config['database'])
    bp.context['db_session'] = get_db_session(engine=bp.context['db_engine'], scope_
↪func=bp.get_current_request)

@bp.event('shutdown')
async def on_shutdown():
    await bp.context['db_engine'].dispose()

@bp.event('request_end')
async def on_request_end():
    await bp.context['db_session'].remove()
```

And then we use these resources inside the project:

```
@routes.get(r'^/hello-world$')
async def hello(request):
    db_session = request.app.context['db_session']()
    ...
```

In this example, we used the `startup` event to initialize the engine and connect to the database at the start of the service, the `request_end` event to remove the dedicated database session of each request at the end of it, and the `shutdown` event to close the database connection when the service shuts down.

Depending on your architecture for managing database connections and sessions, you may want to make the scope of each database session dependent on anything like threads and so on. In this example, the database sessions are set based on the scope of each request, which means that when a request starts, a database session starts (if requested inside the handler by calling db_session) and closes at the end of the request. The Backendpy framework provides the `get_current_request` as a callable for specifying session scope, which can be set in your engine or ORM settings.

Note that in the example above, the names of some functions such as get_db_engine, etc. are used, which have only the aspect of an example and must be implemented by the developer according to the database system used. For more information, you can refer to the specific engine, database or ORM guides you use.

4.10.2 Use Sqlalchemy helper layer

When using Sqlalchemy ORM, Backendpy provides default helpers for this package, which makes it easier to work with.

Note: Async capability has been added from Sqlalchemy version 1.4.27, so lower versions are not compatible with Backendpy framework. Also, among the available engines, only those that support async are usable, such as `asyncpg` package, which can be used based on the `Postgresql` database system.

To use Sqlalchemy in Backendpy projects, do the following:

First, in order to set the database engine and session settings into the project, we use the helper function `set_database_hooks()` as follows:

Listing 27: project/main.py

```
from backendpy import Backendpy
from backendpy.db import set_database_hooks

bp = Backendpy()
set_database_hooks(bp)
```

In addition to setting up the engine and creating and deleting the database connection at the start and shutdown of the service, this function also sets database sessions for the scope of each request, which can be used by calling `request.app.context['db_session']` inside the request handler:

```
@routes.get(r'^/hello-world$')
async def hello(request):
    db_session = request.app.context['db_session']()
    ...
```

The database settings should also be stored in the config.ini file as follows, and the framework will use these settings to connect to the database:

Listing 28: project/config.ini

```
[database]
host = localhost
port = 5432
name = your_db_name
username = your_db_user
password = your_db_password
```

After setting up the project, here's how to use Sqlalchemy ORM in applications:

To create models of an application, inside the desired module of the application, we use the `Base` class as follows:

Listing 29: project/apps/hello/db/models.py

```
from sqlalchemy import Column, Integer, String
from backendpy.db import Base

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer(), primary_key=True)
    first_name = Column(String(50))
    last_name = Column(String(50))
```

If you use this Base class, it is possible to connect between models of different applications, and also the CLI commands of the framework related to the database can be used.

After defining the data models, these models should also be introduced to the application (so that they can be imported when needed for the framework). For this purpose, according to the procedure of other sections, we will use `main.py` module of the application:

Listing 30: project/apps/hello/main.py

```
from backendpy.app import App

app = App(
    ...
    models=['project.apps.hello.db.models'],
    ...)
```

As shown in the example, to introduce the models, we set their module path as a string to the application `models` parameter. This parameter is of iterable type and several model modules can be assigned to it. These module paths must also be within valid Python path. In this example, it is inside the project path that has already been added to the Python path by default.

We can now use database queries in any part of the application:

Listing 31: project/apps/hello/db/queries.py

```
from .models import User

async def get_user(session, identifier):
    return await session.get(User, identifier)
```

Listing 32: project/apps/hello/controllers/handlers.py

```
...
from ..db import queries

@routes.get(r'^/users/(?P<identifier>.+)$', data_handler=UserFilterData)
async def user_detail(request):
    data = request.cleaned_data
    db_session = request.app.context['db_session']()
    result = await queries.get_user(db_session, data['identifier'])
    return Success(to_dict(result))
```

Note that in the sample code above, some functions such as to_dict or UserFilterData, etc. are used, which have an example aspect and must be created by the developer.

For more information about Sqlalchemy and how to use it, you can refer to its specific documentation.

Create database and models with command line

If you use the default Sqlalchemy layer as described above, you can automatically create the database and all data models within the project after entering the project path in the command line and using the following command:

```
$ backendpy create_db
```

4.11 Templates

The need to render templates on the server side is used in some projects. External template engines that support async can be used for this purpose. You can also use the framework helper layer for this, which is a layer for using the Jinja2 template engine package. The Backendpy framework facilitates the use of templates with this template engine and adapts it to its architecture with things like template files async reading from predefined application template paths.

An example of rendering a web page template and returning it as a response is as follows.

First we need to specify the application template dirs inside the application `main.py` module with the `template_dirs` parameter of the App class:

Listing 33: project/apps/hello/main.py

```
from backendpy.app import App

app = App(
    ...
    template_dirs=['templates'],
    ...)
```

Then we create the desired templates in the defined path:

Listing 34: project/apps/hello/templates/home.html

```
<!DOCTYPE html>
<html>
<head>
```

(continues on next page)

(continued from previous page)

```
    <title>Backendpy</title>
</head>
<body>
  <h1>{{ message }}</h1>
</body>
</html>
```

Refer to the Jinja2 package documentation to learn the templates syntax.

Finally, we use these template inside a handler:

Listing 35: project/apps/hello/controllers/handlers.py

```
from backendpy.router import Routes
from backendpy.response import HTML
from backendpy.templating import Template

routes = Routes()

@routes.get(r'^/home$')
async def home(request):
    context = {'message': 'Hello World!'}
    return HTML(await Template('home.html').render(context))
```

In this example code, we first initialize `Template` class with the template name and then with the `render` method we render the context values in it (note that this method must also be called async) and then we return the final content with `HTML` response.

Also here you just need to enter the template name and the framework will automatically search for this name in the application template dirs.

Details of the `Template` class are as follows:

4.12 Logging

The Backendpy framework provides a logging class that uses Python standard logging module and differs in the color display of the logs in the command line, which increases the readability of the logs in this environment.

This module has `get_logger()` function with the following specifications:

Example:

```
from backendpy import logging

LOGGER = logging.get_logger(__name__)

LOGGER.debug("Example debug log")
LOGGER.error("Example error log")
```

4.13 Testing

Because of the use of the async architecture in the Backendpy framework, we will need to run the tests as async for most sections in our applications. Hence, the Backendpy framework provides the `AsyncTestCase` class, which is the subclass of `unittest.TestCase`, to which async execution has been added.

Example of testing a database query:

Listing 36: project/apps/hello/tests/test_db.py

```
import unittest
from asyncio import current_task
from backendpy.unittest import AsyncTestCase
from backendpy.db import get_db_engine, get_db_session
from ..db import queries


DATABASE = {'host': 'localhost', 'port': '5432', 'name': 'your_db_name',
            'username': 'your_db_user', 'password': 'your_db_password'}

class QueriesTestCase(AsyncTestCase):

    def setUp(self) -> None:
        self.db_engine = get_db_engine(DATABASE, echo=True)
        self.db_session = get_db_session(self.db_engine, scope_func=current_task)

    async def tearDown(self) -> None:
        await self.db_session.remove()
        await self.db_engine.dispose()

    async def test_get_users(self):
        result = await queries.get_users(self.db_session())
        self.assertNotEqual(result, False)

if __name__ == '__main__':
    unittest.main()
```

API test example:

Listing 37: project/apps/hello/tests/test_api.py

```
import unittest
from backendpy.unittest import AsyncTestCase
from backendpy.utils import http

class MyTestCase(AsyncTestCase):
    async def setUp(self) -> None:
        self.client = http.AsyncHttpClient()

    async def test_user_creation(self):
        async with self.client as http_session:
            data = {'first_name': 'Jalil',
                    'last_name': 'Hamdollahi Oskouei',
                    'username': 'my_user',
```

(continues on next page)

(continued from previous page)

```
                  'password': 'my_pass'}
            result = await http_session.post('http://127.0.0.1:5000/users', json=data)
            self.assertEqual(result.get('status'), 'success')

if __name__ == '__main__':
    unittest.main()
```

4.14 Initialization scripts

Some applications require basic data in order to run. For example, the users application, after installation, also needs to enter the administrator account information to be able to manage other accounts and the entire system. To record this raw data, we can use initialization scripts that are executable at the command line when a project is deployed.

From the `init_func` parameter of any application we can assign an initialization function to it:

Listing 38: project/apps/hello/main.py

```
from backendpy.app import App
from .controllers.init import init_data

app = App(
    ...
    init_func=init_data,
    ...)
```

And an init function could be like this:

Listing 39: project/apps/hello/controllers/init.py

```
from asyncio import current_task
from backendpy.db import get_db_engine, get_db_session
from backendpy.logging import logging
from ..db import queries

LOGGER = logging.getLogger(__name__)

async def init_data(config):
    # Get db session
    db_engine = get_db_engine(config['database'], echo=False)
    db_session = get_db_session(db_engine, current_task)

    try:
        # ...
        # Create admin user
        if await queries.get_users(db_session):
            LOGGER.warning('Users created already [SKIPPED]')
        else:
            try:
                user_data = {
                    'first_name': input('Enter first name:\n'),
                    'last_name': input('Enter last name:\n'),
```

(continues on next page)

(continued from previous page)

```
                'username': input('Enter admin username:\n'),
                'password': input('Enter admin password:\n')}
        except Exception as e:
            raise Exception(f'Input error:\n{e}')

        if await queries.set_user(db_session, user_data):
            LOGGER.info('Admin user created successfully')
        else:
            raise Exception('Admin user creation error')
    finally:
        await db_session.remove()
        await db_engine.dispose()
```

As can be seen, the init functions receive `config` parameter, which can be used to access project configurations such as database information and so on.

The project manager can perform the initialization by executing the following command on the command line in the project dir:

```
$ backendpy init_project
```

By executing this command, the Backendpy framework executes the initialization scripts of all applications activated in the project configuration sequentially.

CHAPTER

FIVE

PROJECT DEPLOYMENT

A project based on the Backendpy framework is a standard ASGI application and can use a variety of methods and tools to deploy and operate these types of applications.

Web servers such as Uvicorn, Hypercorn and Daphne can be used for this purpose. Also, the features of a web server such as Gunicorn can be used in combination with previous web servers. Or even use them behind the Nginx web server (as a proxy layer) and take advantage of all the features of this web server. To use each of these web servers, refer to their documentation.

Example of using Uvicorn:

```
uvicorn main:bp --host '127.0.0.1' --port 8000
```

Example of using Uvicorn with Gunicorn:

```
gunicorn main:bp -w 4 -k uvicorn.workers.UvicornWorker
```

These commands can be defined and managed as a service in the operating system.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search